MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo 473

May 31, 1978

A Three Valued Truth Maintenance System

David A. McAllester

Abstract

Truth maintenance systems have been used in recently developed problem solving systems. A truth maintenance system (TMS) is designed to be used by deductive systems to maintain the logical relations among the beliefs which those systems manipulate. These relations are used to incrementally modify the belief structure when premises are changed, giving a more flexible context mechanism than has been present in earlier artificial intelligence systems. The relations among beliefs can also be used to directly trace the source of contradictions or failures, resulting in far more efficient backtracking.

In this paper a new approach is taken to truth maintenance algorithms. Each belief, or proposition, can be in any one of three truth states, true, false, or unknown. The relations among propositions are represented in disjunctive clauses. By representing an implication in a clause the same algorithm that is used to deduce its consequent can be used to deduce the negation of antecedents that would lead to contradictions. A simple approach is also taken to the handling of assumptions and backtracking which does not involve the non-monotonic dependency structures present in other truth maintenance systems.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research was provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643 and in part by the National Science Foundation under grant MCS77-04828.

Acknowledgements

Jon Doyle and Gerald Sussman have provided encouragement and indespensible criticism of this work. Johan de Kleer, Charles Rich, and Howard Shrobe have also helped with fruitful discussions.

Table of Contents

Introduction

Truth maintenance systems have been used in recently developed problem solving systems [Stallman and Sussman 1976] [Doyle 1978a]. A truth maintenance system (TMS) is a domain independent system for maintaining the consistency and well foundedness of a set of beliefs. It is an inherenlty propositional mechanism which is designed to be used by deductive systems to maintain the logical relations between the propositions they generate. The truth maintenance system also simulates the effects of those relations to the extent that it can be used to incrementally modify the belief structure and retract assumptions when they lead to contradictions. This process can be used to realize substantial search efficiencies in that contradictions, or failures, only result in backtracking over relevant assumptions.

An example of the use of a TMS would be an algebraic manipulator which is capable of using piecewise approximations to functions. In such a manipulator each equation is considered to be a proposition. When the manipulation of some set of equations results in a new equation, the equations used are recorded in the TMS as supporting the newly generated equation. A piecewise linear approximation to a function can be represented by an implication between equations such as (> x -.25) ∧ (< x .25) → (= (sin x) x). To use such an approximation of (sin x) the manipulator might have to assume that x was in the required range. If at some later time a value for x is found that is inconsistent with the assumptions, then the manipulator need simply announce to the TMS that the two conflicting equations, the assumed inequallity and the found value, are mutually contradictory. The TMS will then retract a relevant assumption. In such cases the TMS might instead be made to state all the assumptions upon which the contradiction depends and leave the choice of removal up to the manipulator. The use of piecewise linear approximations to the behavior of transistors is one of the applications of truth maintenance used by Stallman and Sussman in electronic circuit analysis [Stallman and Sussman 1976].

In addition to the search efficiencies which are gained in dependency directed backtracking, the recorded relations between beliefs can be used to justify or explain the beliefs of the deductive system. Such explanations are useful in understanding and verifying the results of problem solvers.

This paper introduces a new approach to the concepts and mechanisms of truth maintenance. The basic idea is to represent all logical relations between proposition in disjunctive clauses. For example implications of the form P1 ∧ P2 ∧ P3 ... → Q will be represented as ¬P1 ∨ ¬P2 ∨ ¬P3 ... ∨ Q. Notice that in the clause representation the distinction between antecedents and consequences disappears and therefore the negation of an antecedent in the implication can be as easily deduced as the consequent. This feature of the representation eliminates much of the backtracking mechanisms which are present in other truth maintenance systems [Stallman and Sussman 1976] [Doyle 1978a]. Another common relation among propositions

we wish to be able to express is the notion that some set of them are mutually contradictory, formally ¬(P1 ∧ P2 ∧ P3 ... ∧ Pn). This relation is transformed via DeMorgan's laws into the clause ¬P1 ∨ ¬P2 ∨ ¬P3 ... ∨ ¬Pn. This again eliminates the need for certain backtracking mechanisms present in other systems.

The propositions in other systems have only two truth states called "in" and "out", which represent "known to be true" (a well founded proof exists) and "not known to be true" (not in) respectively. My system uses the three more intuitive truth states of true, false, and unknown (hence the title of this paper). This eliminates the need of a separate entity to represent the negation of a proposition.

The notion that the truth of some proposition is an assumption is simply represented by appropriately tagging the proposition. When the assumption is found to lead to a contradiction (a clause that cannot be satisfied) the truth value is automatically retracted. I believe that this mechanism has most of the non-monotonic power of Doyle's system, but in a much simpler form (see [Doyle 1978a] for a discussion of non-monotinicity).

At the end of the paper are a series of appendices which give the details of the implementation and an overview of its use.

The Algorithm

The basic truth maintenance system (TMS) object types are literals (TMS nodes), truth values, terms, and clauses. TMS nodes represent assertions of the deductive system using the TMS. Such assertions might be of the form (COLOR A RED) or (MODE TRANSISTOR-1 BETA-INFINITE), but their TMS representation is simply a unique atom, i.e. a node. Nodes can have three possible truth states, true, false, and unknown. A truth value is a true or false value of a node. Changing a node from an unknown state to either true or false will be referred to as adding a truth value, since it is conceptually adding information. Changing a node from true or false to unknown will be referred to as removing a truth value, as it is conceptually removing information. A term is an association of a node with a value and is true when the node has that value, false when the node has the opposite value, and unknown otherwise.

The relations between the truth values of the nodes are represented by propositional formulas in conjunctive normal form. This means that there is a set of disjunctive clauses which must all be satisfied by the values of the nodes. Each clause contains a set of terms, one of which must be true.

Justifications for assertions are represented as clauses. For example, if an assertion represented by the node C was implied by assertions represented by A and B, then the clause would be ((A.false) or (B.false) or (C.true)). The fact that some set of assertions are mutually contradictory is also represented in a clause. For example, if assertions represented by A,B, and C would lead to a contradiction, then the clause would be ((A.false) or (B.false) or (C.false)).

Each clause can be given multiple interpretations. For example the clause ((A.false) or (C.false) or (D.true)) might be thought of as (A and C) -> D, or it could be thought of as (¬D and C) -> ¬A. This clause can also be thought of expressing the fact that a contradiction results from A, C, and ¬D all being true simultaneously. Even more bizarre conceptualizations of the clause are possible, such as (A -> (¬D -> ¬C)). Interpretations of the last type are useful in understanding certain backtracking techniques to be discussed later.

Adding Clauses and Truth Values

Clauses can be directly added to system at any time by a top level procedure and are instantly checked for possible deductions. Truth values can be added in two ways. The simplest is to add a truth value for a node as a premise. In this case no other reason for believing the value is needed. The second way is to deduce a truth value from a clause. Suppose all the terms of a clause are known to be false with one exception, which is a node whose truth value is unknown. In this case the one remaining node can have the appropriate truth value added to satisfy the clause. This is the only way truth values are deduced from the clauses in the TMS. There are however valid deductions which depend on more than one clause. For example given the two clauses A -> B and ¬A -> B it is possible to

deduce B. Such deductions are only made indirectly when certain types of contradictions arise (contradictions will be discussed in later sections).

When a truth value is added a check must be made to see if new truth values can be deduced from the added information. This is done by examining clauses which contain the term whose truth value has been added. Since clauses which contain the term which is made true are automatically satisfied, the only clauses that must be checked for possible deductions are those that contain the term which is made false. Since truth values are added recursively, all truth values that can be deduced via chains of such one-step clause deductions are added.

For reading the following code it will be useful to refer to Appendix II in which the data structures are explained. The code presented below is a slight simplification of the actual code used here only to formalize the algorithm as described so far. It does not contain the mechanisms for handling contradictions which will be explained later. The complete code is given in Appendix III.

```
(DEFUN SET-TRUTH (NODE VALUE SUPPORT)

        ;SUPPORT IS EITHER THE ATOM 'PREMISE OR
        ;A CLAUSE WHICH IS BEING USED TO DEDUCE VALUE

       (PROG ()
             (COND ((NOT (EQ (GET NODE 'TRUTH) 'UNKNOWN))
                    (ERROR 'SET-TRUTH--VALUE-NOT-UNKNOWN NODE)))
             (PUTPROP NODE VALUE 'TRUTH)
             (PUTPROP NODE SUPPORT 'SUPPORT)

               ;FOR EACH CLAUSE WHICH CONTAINS THE TERM WHICH BECOMES FALSE
               ;SUBTRACT ONE FROM THE NUMBER OF TERMS WHICH CAN POTENTIALLY SATISFY IT.

             (MAPC (FUNCTION (LAMBDA (CLAUSE)
                             (PUTPROP CLAUSE (1- (GET CLAUSE 'PSAT)) 'PSAT)))
                   (GET NODE (GET VALUE 'OP-CLAUSES)))

             (MAPC (FUNCTION DEDUCE-CHECK)
                   (GET NODE (GET VALUE 'OP-CLAUSES)))))

(DEFUN DEDUCE-CHECK (CLAUSE)
      (PROG (F)
            (COND ((AND (= (GET CLAUSE 'PSAT) 1)
                        (SETQ F (PCONSEQ CLAUSE)))
                   (SET-TRUTH (CAR F) (CDR F) CLAUSE)))))

        ;PCONSEQ FINDS A NODE IN THE CLAUSE WHICH HAS A TRUTH STATE
        ;OF UNKNOWN AND RETURNS A DOTTED PAIR OF THE NODE AND THE
        ;VALUE WHICH THE NODE MUST HAVE TO SATISFY THE CLAUSE.
```

It would be possible to check for more complex deductions. For example if there are two clauses (P or Q) and (P or ¬Q) it is valid to deduce P. In general, arbitrary deductions could be done by deciding

whether the addition of some truth value inevitably leads to a contradiction. If this is indeed the case then the opposite truth value could be deduced. The problem with this seemingly straightforward approach is in deciding if something must lead to a contradiction. A contradiction is inevitable when the set of clauses can not be satisfied by any truth values for the nodes which are unknown. Therefore in order to decide if a contradiction is inevitable the system must decide if the set of clauses can be satisfied by the remaining unknown nodes. This is a standard problem of propositional logic and is known to be NP complete. This means that there are strong suspicions that it must require exponential time to solve. Therefore, in order to avoid such a combinatorial explosion I restrict myself to one clause deductions. This still gives all of the intuitive deductive power of a clause while preserving computational expedience.

Removing Truth Values

Truth values can be removed as well as added. This can happen when the user of the TMS decides that a premise is no longer known, or it can happen when assumptions are retracted in backtracking. When this happens it is necessary to remove all truth values that critically depend on the lost information. Truth values are used for deductions only by clauses that contain the term they make false. Therefore clauses which contain the term which was previously false, but is now unknown, are examined. If any of these clauses were used in the original deduction of some truth value, then the value deduced is a candidate for removal.

In order to determine whether a clause was the one originally used to deduce a truth value, each node has associated with it a support. The support is only used when the node has a known truth value, and is either a premise marker or the clause which was used in the original deduction of the truth value. Since the support is always assigned when a truth value is added, the truth values of the other nodes in the support can in no way depend on the supported value. This means that the support is well founded and the set of premises that a truth value is deduced from can always be determined by tracing supports without fear of loops.

Care must be taken that values are not removed that can be deduced in other ways. One attempt at solving this problem is to check all clauses that contain the node whose value is being considered for removal to see if any can be used to deduce the value. However the following example demonstrates the problem with this approach. Consider the clauses:

A -> B

B -> C

C -> B

Suppose that A was added as a premise and then later removed. Now when A is removed there is still a clause, C -> B, that can be used to deduce B. The problem with using this clause to support the truth of B is that, since C depend on B, B would be used to support itself. The solution to this problem is to first recursively remove all candidates for removal (therefore removing all truth values that critically depend on them). After this has been done clauses which contain the terms whose values have been removed can be checked for deductions. If contradictions are present in the system, then it is possible that when a truth value is removed its opposite value can then be deduced (contradictions will be discussed in more detail a little later). Whenever a clause is used to deduce a truth value, the clause becomes the support for the value. Again since the support is found before the truth value is added, it must be well founded. An important point is that if a premise is removed by the TMS user, but the removed truth value can be deduced from other premises in the system, then the truth value remains, with a clause as its support instead of the premise marker.

In reading the following code it will again be helpful to refer to Appendix II in which the data structures are explained. The code presented here is a simplification of the actual code used only to formalize the algorithm as discussed so far, i.e. it does not deal with contradictions and assumptions which will be discussed later. The complete code is presented in Appendix III.

```
(DEFUN REMOVE-TRUTH (NODE)
       (PROG (VALUE)
             (SETQ VALUE (GET NODE 'TRUTH))
             (COND ((EQ VALUE 'UNKNOWN)
                    (ERROR 'REMOVE-TRUTH--VALUE-NOT-PRESENT NODE)))
             (PUTPROP NODE 'UNKNOWN 'TRUTH)
             (PUTPROP NODE NIL 'SUPPORT)

                ;FOR EACH CLAUSE WHICH CONTAINS THE TERM WHICH WAS FALSE
                ;ADD ONE TO THE NUMBER OF TERMS WHICH CAN POTENTIALLY SATISFY IT

             (MAPC (FUNCTION (LAMBDA (CLAUSE)
                             (PUTPROP CLAUSE 'PSAT (1+ (GET CLAUSE 'PSAT))) ))
                (GET NODE (GET VALUE 'OP-CLAUSES)))

          ;REMOVE TRUTH VALUES WHICH THESE CLAUSES HAD BEEN USED TO DEDUCE

       (MAPC (FUNCTION (LAMBDA (CLAUSE)
                (PROG (F)
                   (COND ((AND (> (GET CLAUSE 'PSAT) 1)
                               (SETQ F (CAR (CONSEQ CLAUSE)))
                                   ;CONSEQ FINDS A NODE WHICH SATISFIES THE CLAUSE.
                               (EQ CLAUSE (GET F 'SUPPORT)))
                          (REMOVE-TRUTH F))))))
             (GET NODE (GET VALUE 'OP-CLAUSES)))

          ;CHECK FOR ANY POSSIBLE DEDUCTIONS OF VALUES FOR THE
          ;NODE WHOSE VALUE WAS REMOVED

       (MAPC (FUNCTION DEDUCE-CHECK) (GET NODE 'POS-CLAUSES))
       (MAPC (FUNCTION DEDUCE-CHECK) (GET NODE 'NEG-CLAUSES)) ))
```

Contradictions

Consider a case in which a clause is added that contains only terms which are false. The clause is in contradiction with the rest of the data base and is therefore referred to as a contradiction. Since a clause is a contradiction only when all of the terms in it are false, a contradiction is said to depend on the truth values of the terms in it. It is conceivable that a TMS data base could contain several such contradictions.

The addition of clauses is not the only way that contradictions can occur. Consider the two implications $(P \rightarrow Q)$ and $(P \rightarrow \neg Q)$. If no truth values are known for P or Q, then no deductions are made since each clause has two ways in which it might be satisfied. If a true value for P is determined, then one of the above clauses would be used to deduce a truth value for Q, while the other clause would become a contradiction. It is important to realize that in cases where both a

truth value and its negation can be proven, one of the truth values is chosen and all clauses which could have implied its negation become contradictions. When adding a truth value leads to a contradiction, it is possible to add new clauses that allow deductions based upon this fact. In the above example the clause (¬P) can be deduced from the two original clauses. To get a better feel for the general case consider the example:

clause	interpretation
((A. true) (B. false) (C. true))	(¬A ∧ B) → C
((C. false) (D. false) (E. true))	(C ∧ D) → E
((A. true) (F. false) (E. false))	(¬A ∧ F) → ¬E

known values

B true

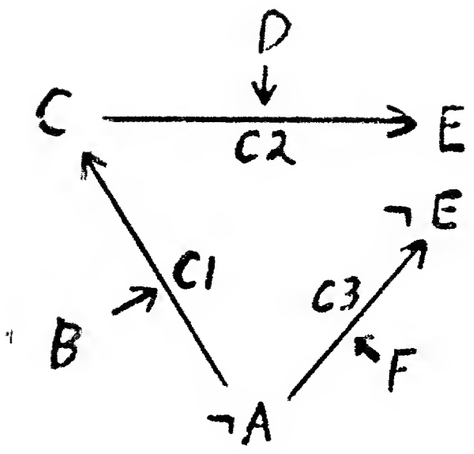
D true

F true

Now in this case if a false value for A is added the first clause can be used to deduce C. Then the second clause can be used to deduce E. At this point however the third clause has become a contradiction. The relationship between the three clauses is shown in figure one. In the figure each clause is represented by a pair of right angled implication pointers which should be interpreted as B → (¬A → C). The new clause which can be added in this situation is (B ∧ D ∧ F) → A

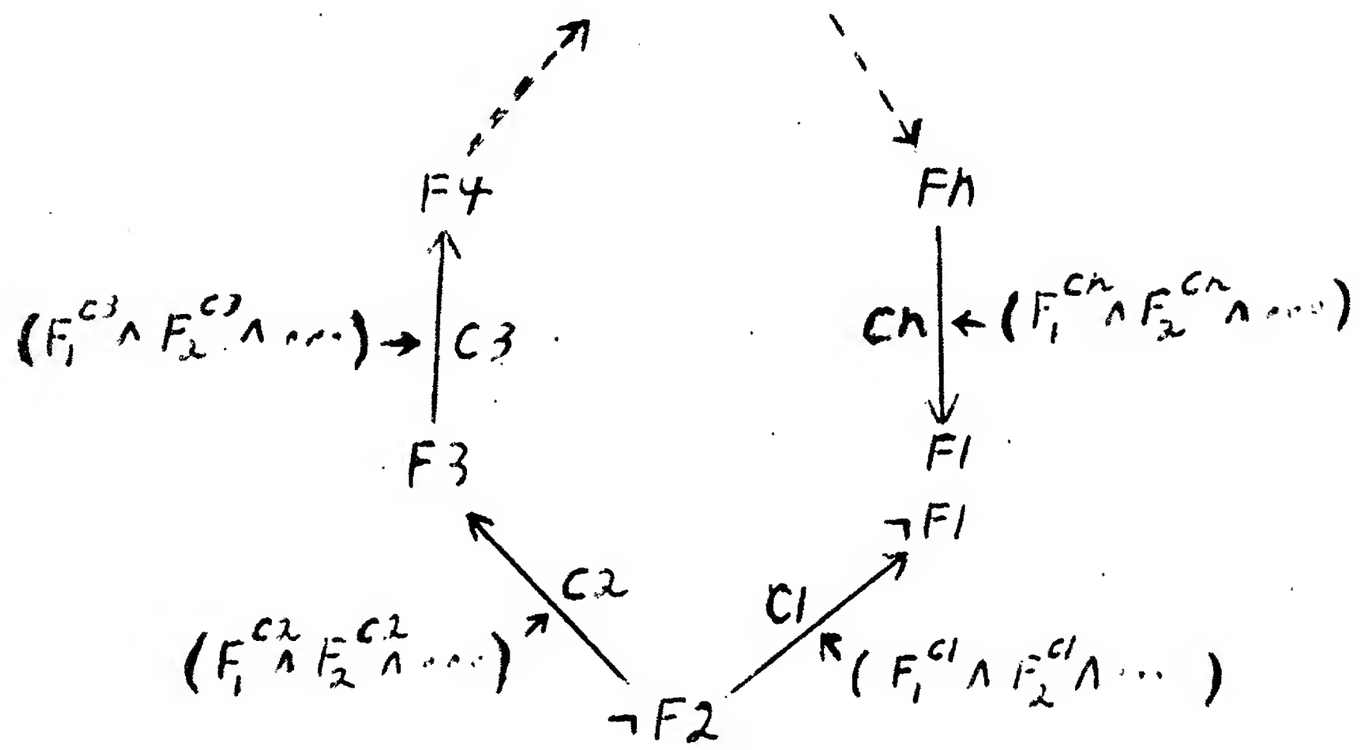
To see how new clauses can be constructed from the appearance of contradictions in general it is necessary to closely examine how contradictions result from the addition of truth values. At some point a truth value is added which removes the last chance of satisfying some clause, say C1. The term that became true when this value was added will be called F1 (E in the above example). At the instant before this truth value is added C1 could have been used to deduce the opposite value. In a quiescent data base no such clause can exist since all possible one step deductions are made. This means that a false truth value was added for some term in C1 (A in the above example), but that C1 became a contradiction before it could be checked for deductions. This second term will be called F2. All possible deductions from the addition of any truth value are made when the value is added, and C1 could potentially have been used to deduce ¬F1 upon the addition of ¬F2. Therefore the truth value of F1, which caused C1 to be a contradiction, must also be a deduction from ¬F2. This situation is pictured in figure two.

In the figure each pair of right angled implication pointers represents a clause. The clauses should be interpreted as $((F^{C1}_1 \wedge F^{C1}_2 \wedge \ldots) \rightarrow (\neg F2 \rightarrow \neg F1))$. Now if all the involved clauses



$$(B \wedge D \wedge F) \rightarrow A$$

Figure 1.
An example of clause formation as a result of a contradiction. The addition of ¬A causes C3 to become a contradiction



$$\left((F_1^{C1} \wedge F_2^{C1} \wedge \cdots) \wedge (F_1^{C2} \wedge F_2^{C2} \wedge \cdots) \wedge \cdots \right) \longrightarrow F2$$

Figure 2.
The general case of clause formation resulting from a contradiction. This is the general case in which the addition of ¬F2 causes C1 to become a contradiction.

contained only two terms, then the clause (F2) could be added. However, in the general case the contradiction results only when the peripheral terms are true. In other words ($F^{C1}_1 \wedge F^{C1}_2 \wedge \ldots F^{C2}_1 \wedge F^{C2}_2 \wedge \ldots.$) -> F2. This implication then is the clause which is added. The clause is formed during the unrolling of the recursive calls to the procedure for adding truth values.

Hopefully the added clause will allow the deduction of the negation of the truth value that lead to the contrdiction, ¬F2 above, in those cases when it would again lead to the same contradiction. While the added clause is always valid, it does not always produce this desired result. The reason for this is that some of the peripheral truth values might also be deductions from the added value. In this case when the added value (call it ¬F2 as above) is retracted, some of the peripheral values will also disappear and its negation will not be deducible. However if ¬F2 is again added then the clause generated above will become a contradiction at the point at which all the peripheral values become true. This will lead to the generation of yet another clause. If F2 is still not deducible upon the retraction of ¬F2, then further additions of ¬F2 will generate still more clauses. It is always possible to force the system to deduce the negation of a truth value that leads to a contradiction by such "pulsing" of that value. I would like to emphasize that it would require a quite complex structure to require more than one or two such "pulses".

Default Values and Backtracking

In many problem solving situations it is necessary to make assumptions that have no solid reason for belief. If such assumptions lead to contradictions then they should be retracted. Assumptions are represented as a subset of the premise values called default values and are marked in the implementation by having the atom 'default as their support. Whenever a node is given a default support the value supported is placed under a default property of the node. This value is then added whenever no other truth value for the node can be deduced.

When a contradiction is present in the data base an attempt is made to remove it by removing default truth values. This involves tracing the dependency relations (via the supports associated with nodes) to find the premises upon which the contradiction depends and is therefore called dependency directed backtracking. When a default value is found upon which the contradiction depends, it is removed. Hopefully the contradiction becomes an implication and can then be used to deduce the opposite of the default value removed. If the opposite of the default is not deduced, then the default value is added back. At this point, since the contradiction must reappear, the backtracking repeats and again the default value is removed. Due to the new clauses generated each time the contradiction appears, the negation of the default value must become deducible, and the backtracking halts when

either no contradictions are left, or the contradictions that are present do not depend on any default values.

Clause Values and Hierarchies of Assumptions

So far there has been no mention of the removal of clauses from the data base. While the physical removal of clauses does not occur, there is a mechanism for making them impotent. This is done by adding a node to each clause which represents its validity. For example the clause (A or B) might become (¬C1 or A or B) where C1 represents the validity of the clause. Now as long as C1 is true the clause acts as expected, but if the truth value C1 is removed, then the clause is effectively removed. Each time a clause is added, some reason is given for believing it. This reason is used as the support for the truth of the node which represents the clause. This is usually useful only as a device for keeping track of the source of clauses for the TMS user. However it has one very important use in allowing assumptions to have antecedents.

Suppose that in reasoning about animals it is first assumed that they are mammals. Furthermore suppose that in reasoning about mammals it is assumed that they are dogs. The assumption that some animal is a dog might depend on the assumption that it is a mammal. In general then assumptions must be able to take antecedents, some of which might be other assumptions. Dependencies of this form can be represented in the TMS by a clause whose clause node has a default support. In the example let the mammal assumption be represented by A1 and the dog assumption by A2. Now A1 can be assumed (given a true default value). Once this has been done A2 can be added by adding the clause (A1 -> A2) and giving the clause node a true default value. The clause allows the deduction of A2 only if A1 is believed, also A2 can be removed as an assumption during backtracking by removing the default truth value of the added clause.

Since clause nodes are no different than any other nodes, removal of their default truth values could be taken care of by the backtracking algorithm already described. However, in backtracking it is desirable to remove first assumptions upon which no other assumptions depend. In chess for example one normally considers several responses to a given move before going on to the next move. This helps prevent the thrashing involved in removing and adding many assumptions at once.

A minor modification to the backtracking algorithm allows this selection of the default values. The number of supporting clauses that must be traced back from the contradiction to find a premise are counted. There may be several paths of supports that lead to a given premise, and in this case the maximum distance is used. Now if some default value implies some other assumption, then it will always appear at a greater distance from the contradiction than the assumption it

implies. This is because it will always appear in the support chains beneath the implied assumption. In view of this fact, our goal is achieved by choosing for removal the default value with the minimum maximum distance from the contradiction.

Comparison with Other Work

There are several systems which use explicit justifications for beliefs. The work which is most closely related to the TMS presented here is that of Stallman and Sussman [Stallman and Sussman 1976] and Doyle [Doyle 1978a]. The basic difference between these systems and my TMS is in the mechanisms used for dependency directed backtracking. In both of these systems each assertion has only two truth states, believed and unknown, called "in" and "out" respectively. Since no assertion can be false in such a system, additional mechanisms are needed to prevent the belief of sets of assumptions known to be contradictory. In Stallman and Sussman's ARS system the assumptions underlying a contradiction are placed in a NOGOOD assertion. This is used by additional mechanisms which prevent the set of assumptions from being believed.

Jon Doyle has completed a master's thesis on the implementation of a general purpose truth maintenance system (TMS) [Doyle 1978a]. His TMS employs the notions of "in" and "out" to represent the truth values of assertions. If an assertion is believed by the system then its TMS node is "in". If the assertion is not believed, i.e. either known to be false or simply unknown, then its TMS node is "out". To make a distinction between simply not knowing something's truth value and knowing that it is false, two TMS nodes are required, one for the assertion and one for its negation. If the assertion has an unknown truth value, then both nodes are "out". If the truth value is known then one of the nodes is "in" and the other is "out". An important point is that in Doyles system the notion of a contradiction is completely seperate from the justifications. A node must be created which is declared to be a contradiction and then a justification of this contradiction node is installed for each set of nodes which are mutaully contradictory. Specifically a contrdiction node must be implied by each pair of nodes representing an assertion and its negation. If this is not done then the system will have no problem with keeping both of these nodes in, in other words believing both an assertion and its negation.

In Doyle's system a mechanism of major importance in backtracking is conditional proof. A conditional proof can be thought of as a TMS node which is true whenever a certain implication among other TMS nodes is true. Therefore a conditional proof is defined in terms of the implication it represents, in other words as a set of antecedents and a consequent. To take an example suppose that the implication $A \wedge B \wedge C \rightarrow D$ is present in some TMS. A conditional proof node of D with respect to A and B would represent the truth of the implication $A \wedge B \rightarrow D$. Now if C were known to be true then the implication represented by the conditional proof would be true and therefore also the "CP" node. This can be stated more formally as $C \rightarrow (A \wedge B \rightarrow D)$.

Doyle's TMS uses unidirectional justifications to keep track of consequences and antecedents of beliefs. To achieve dependency directed backtracking in such systems it is necessary to trace the antecedents of a failure (or contradiction) and then to blame the

contradiction on some set of assumptions. Once this is done it is also necessary to add implications for the negations of each assumption to prevent the set of assumptions from being again believed at some later time.

Doyle's TMS also uses a non-monotonic dependency structure. This means that truth of a node can depend on another node being unknown. In such cases whenever the unknown, i.e. "out", node becomes known, i.e."in", the dependant node becomes "out". Whenever an assumption is made, an node representing its negation is created. The assumption is then made to depend on its negation being unknown. This can be understood as expressing the notion "I will believe it to be true as long as I cannot prove that it is false". When a contradiction is found which depends on some set of assumptions a conditional proof is constructed for each assumption representing the implication of the contradiction by the assumption. This conditional proof is then used to imply the negation of each assumption when that assumption would lead to the contradiction. Since an assumption depends on its negation being "out", when the negation becomes "in" the assumption becomes "out".

The implementation of the conditional proof mechanism is however not complete. If the antecedants of the implication a conditional proof represents are not "in", then it is extremely difficult and computational expensive to check the actual truth value of the conditional proof. Doyle's TMS "CP" nodes are "in" in only a subset of the cases in which they could actually be shown true. For example the conditional proofs which result from a contradiction are each associated with the set of assumptions upon which that contradiction was blamed. The negation of one of these assumptions is only justified when all the other assumptions in that specific set are believed. Therefore the conditional proof will not justify the negation of an assumption when the assumption can combine with other information to give the contradiction. New assumptions which lead to old contradictions are also not prevented from being believed. By using the contradiction itself as an active deductive agent, my TMS can make such deductions and avoid backtracking entirely in many such situations.

In my TMS a contradiction is only a clause which cannot be satisfied. Therefore, if a truth value of one of the nodes in the clause is removed, the clause will be used to deduce the opposite truth value for this node. This allows the direct deduction of false values for assumptions which would lead to contradictions. The main advantage of this system is the tremendous conceptual and programming simplicity achieved. However it is more than just a simplified implementation of the previous systems. Because no specific set of assumptions is chosen upon which to blame a contradiction, new assumptions which would lead to the contradiction can also be proven false. However the difference this makes in the number of contradictions encountered is small compared to the savings given by any form of dependency directed backtracking.

I have not implemented any conditional proof mechanism for the

simple reason that I did not have any application to justify its existence. Doyle's TMS uses conditional proof to implement a mechanism for levels of abstraction. This is useful in condensing explanations of beliefs. The exploration of such uses of conditional proof and its relation to the algorithms presented here would be an interesting topic for further research. Also there are probably uses for the non-monotonic dependency structure present in Doyle's TMS other than the identification of assumptions. The incorporation of such mechanisms into this algorithm and the investigation of more sophisticated applications would also be worthwhile.

Bibliography

[de Kleer,Doyle,Rich,Steele and Sussman 1977]
Johan de Kleer, Jon Doyle, Charles Rich, Guy L. STeele Jr., Gerald J. Sussman, "AMORD A Deductive Procedure System," MIT AI Lab, Memo 435, September 1977.

[de Kleer,Doyle,Steele and Sussman 1977]
Johan de Kleer, Jon Doyle, Guy L. STeele Jr., Gerald J. Sussman, "Explicit Control of Reasoning," MIT AI Lab, Memo 427, June 1977.

[Doyle 1978a] Jon Doyle, "Truth Maintenance Systems for Problem Solving," MIT AI Lab, TR-419, January 1978.

[Doyle 1978b]
Jon Doyle, "A Glimpse of Truth Maintenance," MIT AI Lab, Memo 461, February 1978.

[McDermott 1976]
Drew V. McDermott,"Flexibility and Efficiency in a Computer System for Designing Circuits," MIT AI Lab, TR-402,June 1977.

[McDermott and Sussman 1974]
Drew V. Mcdermott and Gerald Sussman, "The CONNIVER Reference Manual," MIT AI Lab, Memo 259a, January 1974.

[Stallman and Sussman 1976]
Richard Stallman and Gerald Sussman, "Forward Reasoning and Dependency Directed Backtracking in a System for Computer Aided Circuit Analysis," MIT AI Lab Memo 380, September 1976.

[Sussman 1977]
Gerald Sussman, "Slices: At the Boundry Between Analysis and Synthesis," MIT AI Lab, Memo 433, July 1977.

Appendix I -- A User's Guide

This is a summary of the top level procedures which can be used to interface the TMS with any deductive system. The procedures descibed here can be loaded into lisp on MIT-AI by evaluating (FASLOAD TMS FASL DSK DAM).

(TMS-INIT)

This procedure initializes the TMS data structures.

(MAKE-DEPENDENCY-NODE <assertion> <when-true> <when-false> <when-unknown>)

This procedure creates and returns a new TMS node to represent the assertion which is passed as an argument. The node returned is an atom which has the assertion placed on its 'ASSERTION property. <when-true> is a function to be applied to the assertion when the node becomes true. <when-false> and <when-unknown> are similarly functions to be used when the node becomes false and unknown respectively. Any of the three function arguments can also be nil in which case no action is taken upon the correspoding transition of truth state.

The truth state of the node is available as its 'TRUTH property which is either 'TRUE, 'FALSE, or 'UNKNOWN. The node is initially assumed unknown and must be forced true via either SET-TRUTH or the addition of a clause which causes its deduction. <when-true> will be applied when this happens. The node can never go directly from being true to being false, or vice-versa, it must first pass through a state of being unknown and therefore an application of <when-unknown>.

(SET-TRUTH <node> <value> <reason>)

This procedure adds truth values. The first argument must be a TMS node and the value must be either 'true or 'false. An error results if the node does not have an initial truth state of unknown. <reason> should be a representation of the reason for believing the truth value and, except for the special atom 'DEFAULT described below, is not used by the TMS other than placing it on the 'EXPLANATION property of the node for the convienience of the TMS user. All truth values which are given via an external call to this procedure are considered premises by the TMS and their 'SUPPORT property is the atom 'PREMISE.

An internal form of this procedure is used to add truth values deduced from clauses. In this case the support is a clause. All truth values that can be deduced in one step clause deductions from any added truth values are also added. If a contradiction results from the addition of a truth value, a new clause is generated and added as described in the section on contradictions

(ADD-CLAUSE <clause> <reason>)

This procedure adds a clause to the TMS data base. The clause argument must be a list of dotted pairs. Each pair is a TMS node dotted with either 'true or 'false to represent the node or its negation respectively. For example:

```
(SETQ A (MAKE-DEPENDENCY-NODE 'HUMAN-TURING nil nil nil))
(SETQ B (MAKE-DEPENDENCY-NODE 'FALLABLE-TURING nil nil nil))
(ADD-CLAUSE (list (cons A 'false ) (cons B 'true)) 'HUMAN-FALLABILLITY)
```

Would add the clause:

(¬CLAUSE-N or ¬HUMAN-TURING or FALLIBLE-TURING)

Which can be more intuitively understood as (HUMAN=FALLABILLITY → (HUMAN-TURING → FALLABLE-TURING)). Clause-n represents the TMS node generated to represent the truth value of the clause and is given a true valus as a premise with an explanation property of <reason> as in SET-TRUTH. The clause node is usefull both in generating explanations and in removing clauses when desired (making the clause node false effectively removes the clause). The clause node is returned as the value of ADD=CLAUSE.

DEFAULTS

The use of the atom 'DEFAULT as the reason argument in either SET-TRUTH or ADD-CLAUSE has a special meaning. It is the way in which assumptions are announced to the TMS. A node or clause given a reason of 'DEFAULT is said to have a default truth value. Such nodes and clauses act just like any other untill contradictions appear. If any contradiction can ever be traced to a node with a default value, then the value for that node will be automatically retracted by the TMS. Similarly if a contradiction can be traced to a deduction from a clause with a default value, then the clause will be invalidated and the deduction retracted. This is the way in which this TMS handles the non-monotonic functions performed in Doyle's TMS [Doyle 1978a].

Notice how default clauses can be used to structure assumptions. Suppose for example that you want to assume that grocery stores have peas. This assumption can be captured in a clause which represents the implication grogery-store → has-peas. The clause would be given a default support to announce that it is an assumption. This is described in more detail in the section on default truth values and hierarchies of assumptions.

If it is desired not to have the TMS do automatic backtracking then the assumptions simply need not be announced to the TMS.

(REMOVE-TRUTH <node>)

This procedure removes the truth value of the node which is passed as an argument. An error results of the node already has an unknown truth state. All truth values which critically depend on the removed value are also automatically removed. At the end of the removal process nodes whoes truth values were removed, but which have default values, have there default values added via SET-TRUTH. It is important to realize that truth values which can be duduced from other knowledge in the TMS will not be removed. Such truth values can only be removed by removing the premises or clauses from which it can be deduced. Clauses are removed by removing the truth value of their TMS nodes.

(WHY <object>)

<object> may be either a node or a clause (more specifically a contradiction, which is a clause which cannot be satisfied). I will first consider the case where <object> is a node. In this case the procedure returns the justification for the belief in the truth value of the node. If the node has truth state of unknown then nil is returned. If the value of the node is a premise, then the atom 'PREMISE is returned. Otherwise it returns a list of the nodes whoes truth values were used to deduce the truth vaue of <node>. The clause node of the supporting clause will appear first on the list. Because of the internally generated clauses which result from the contradictions it is possible that other nodes on this list are clause nodes. Clause nodes can be identified in that there 'ASSERTION property is the atom 'CLAUSE.

If <object> is a clause then Why returns a list of the nodes contained in it. The clause node comes first. It is important to note the distinction between a clause and a clause node. The former represents the actual clauses used by the TMS and it is these which appear on the list of contradictions described below. Clause nodes on the other hand represent the vallidity of a clause and are used for the recording the reasons for belief in clauses and for removing clauses as described above.

WHY can be used to do dependency directed backtracking outside of the TMS in cases where more controle over the choice of assumption removal is desired.

All contradictions that occur in the TMS are placed on a list which is the value of the global atom CONTRA-LIST. If there are no contradictions then the CONTRA-LIST will have value nil. A contradiction is simply a clause which cannot be satisfied.

Appendix II -- The Data Structures

Nodes

Nodes represent assertions or any logical items that take on truth values. Nodes are represented by atoms with the following properties:

TRUTH

This properties can have three values, 'true,'false, and 'unknown, which represent the truth state of the node.

SUPPORT

This property gives the support for a truth value of the node. It is either the atom 'premise or a clause if the node has a truth value, and is nil if the node has a truth property of 'unknown.

POS-CLAUSES

This is the list of clauses which contain the node.

NEG-CLAUSES

This is a list of clauses which contain the negation of the node.

MAKE-TRUE
MAKE-FALSE
MAKE-UNK

These are optional properties which give functions to be applied to the ASSERTION property when the node undergoes the appropriate transition of truth state.

DEFAULT

This applies only to nodes that have default truth values and is either 'TRUE or 'FALSE.

ASSERTION

This is the assertion external to the TMS that the node is representing.

EXPLANATION

This is the reason for believing a node which is a premise, e.g. a node which was set true or false for reasons external to the TMS.

True and False

'True and 'false are atoms with the following properties.

TRUE
OPPOSITE => 'FALSE

```
        CLAUSES => 'POS-CLAUSES
        OP-CLAUSES => 'NEG-CLAUSES
        EFFECT => 'MAKE-TRUE

    FALSE
        OPPOSITE => 'TRUE
        CLAUSES => 'NEG-CLAUSES
        OP-CLAUSES => 'POS-CLAUSES
        EFFECT => 'MAKE-FALSE
```

Clauses

Clauses are atoms with the following properties

CLAUSE-LIST

This is the list structure which contains the nodes and associated truth values which make up the clause. It is a list of dotted pairs each of which is a node dotted with the truth value it has in the clause.

PSAT

This is the number of nodes which either satisfy the clause or could potentially do so. If this number is 1, and there is a node with an unknown truth value in the clause, then the clause can be used to deduce a truth value for the node. If this number is 0 then the clause is a contradiction.

Global Variables

CONTRA-LIST

This is a list of all the clauses which are contradictions.

The following variables are global to certain procedures in the TMS.

CONTRA-CLAUSE

This is used to construct the new clause resulting from the appearance of contradictions. It is global to the internal version of SET-TRUTH.

CONTRA-SOURCE

This is the contradiction which initialized the construction of a new clause. It is used by the internal version of SET-TRUTH to terminate the construction of the new clause.

REMOVED-LIST

This is used by REMOVE-TRUTH to keep track of nodes whose truth values have been removed.

ASSUM-LIST

This is used in FIND-ASSUM for accumulating an alist of the assumptions underlying a contradiction associated with their maximum distance from the contradiction. The distance to the contradiction is the number of clauses in the link between the contradiction and the assumption.

Appendix III -- The Code

```
001   ;INITIALIZATION ROUTINES
002
003   (DECLARE (SPECIAL CONTRA-LIST NODE-COUNT CTRACE VPRINT ACCUM
004                     CONTRA-SOURCE CONTRA-CLAUSE REMOVED-LIST ASSUM-LIST))
005
006   (DEFUN  TMS-INIT ()
007       (PROG ()
008          (PUTPROP 'TRUE 'FALSE 'OPPOSITE)
009          (PUTPROP 'TRUE 'POS-CLAUSES 'CLAUSES)
010          (PUTPROP 'TRUE 'NEG-CLAUSES 'OP-CLAUSES)
011          (PUTPROP 'TRUE 'MAKE-TRUE 'EFFECT)
012          (PUTPROP 'FALSE 'TRUE 'OPPOSITE)
013          (PUTPROP 'FALSE 'NEG-CLAUSES 'CLAUSES)
014          (PUTPROP 'FALSE 'POS-CLAUSES 'OP-CLAUSES)
015          (PUTPROP 'FALSE 'MAKE-FALSE 'EFFECT)
016          (SETQ CONTRA-LIST NIL)
017          (SETQ CTRACE NIL)
018          (SETQ VPRINT NIL)))
019
020
021   (DEFUN MAKE-DEPENDENCY-NODE (ASSERTION WHEN-TRUE WHEN-FALSE WHEN-UNKNOWN)
022          (PROG  (NODE)
023                 (GENSYM 'N)
024                 (SETQ NODE (GENSYM))
025                 (INTERN NODE)
026                 (PUTPROP NODE 'UNKNOWN 'TRUTH)
027                 (PUTPROP NODE NIL 'SUPPORT)
028                 (PUTPROP NODE ASSERTION 'ASSERTION)
029                 (PUTPROP NODE WHEN-TRUE 'MAKE-TRUE)
030                 (PUTPROP NODE WHEN-FALSE 'MAKE-FALSE)
031                 (PUTPROP NODE WHEN-UNKNOWN 'MAKE-UNK)
032                 (RETURN NODE)))
```

```
001
002   ;CLAUSE ADDITION ROUTINES
003
004   (DEFUN ADD-CLAUSE (CLAUSE-LIST REASON)
005          (ADD-2 CLAUSE-LIST REASON)
006          (BACKTRACK))
007
008   (DEFUN ADD-2  (CLAUSE-LIST REASON)
009          (PROG (CLAUSE CLAUSE-NODE COUNT)
010                (SETQ COUNT 0)
011
012                (SETQ CLAUSE-NODE (MAKE-DEPENDENCY-NODE 'CLAUSE NIL NIL NIL))
013                (PUTPROP CLAUSE-NODE 'TRUE 'TRUTH)
014                (PUTPROP CLAUSE-NODE 'PREMISE 'SUPPORT)
015                (PUTPROP CLAUSE-NODE REASON 'EXPLANATION)
016                (COND ((EQ REASON 'DEFAULT)
017                       (PUTPROP CLAUSE-NODE 'TRUE 'DEFAULT)))
018
019                (GENSYM 'C)
020                (SETQ CLAUSE (GENSYM))
021                (INTERN CLAUSE)
022                (COND (CTRACE
023                       (PRINT '|NEW-CLAUSE |)
024                       (PRINC  CLAUSE)
025                       (PRINC '|  |)
026                       (PRINC REASON)
027                       (PRINT (MAPCAR (FUNCTION (LAMBDA (F)
028                                                 (CONS (GET (CAR F) 'ASSERTION) (CDR F))))
029                                      CLAUSE-LIST))))
030
031                (SETQ CLAUSE-LIST (CONS (CONS CLAUSE-NODE 'FALSE) CLAUSE-LIST))
032                (PUTPROP CLAUSE CLAUSE-LIST 'CLAUSE-LIST)
033                (MAPC (FUNCTION (LAMBDA (NODE)
034                                (COND ((NOT (EQ (GET (CDR NODE) 'OPPOSITE)
035                                                (GET (CAR NODE) 'TRUTH)))
036                                       (SETQ COUNT (1+ COUNT))))))
037                      CLAUSE-LIST)
038                (PUTPROP CLAUSE COUNT 'PSAT)
039
040                (MAPC (FUNCTION (LAMBDA (NODE)
041                                (PUTPROP (CAR NODE)
042                                         (CONS CLAUSE
043                                               (GET (CAR NODE) (GET (CDR NODE) 'CLAUSES)))
044                                         (GET (CDR NODE) 'CLAUSES))))
045                      CLAUSE-LIST)
046
047                (COND ((= COUNT 0)
048                       (SETQ CONTRA-LIST (CONS CLAUSE CONTRA-LIST))))
049                (PROG (CONTRA-SOURCE CONTRA-CLAUSE)
050                      (DEDUCE-CHECK CLAUSE))
051                (RETURN CLAUSE-NODE)))
```

```
001
002  ;TRUTH VALUE ADDITION ROUTINES
003
004  (DEFUN SET-TRUTH (NODE VALUE EXPLAN)
005         (PROG (CONTRA-SOURCE CONTRA-CLAUSE)
006               (PUTPROP NODE EXPLAN 'EXPLANATION)
007               (COND ((EQ EXPLAN 'DEFAULT) (PUTPROP NODE VALUE 'DEFAULT)))
008               (SET-2 NODE VALUE 'PREMISE)
009               (BACKTRACK)))
010
011  (DEFUN SET-2 (NODE VALUE SUPPORT)
012         (PROG (TRACE F)
013               (COND (VPRINT (PRINT '|SET TRUTH|) (PRINC NODE) (PRINC VALUE) (PRINC SUPPORT)))
014
015               ;TRACE IS TRUE IF A CONTRADICTION HAS RESULTED FROM THIS SET, EITHER DIRECTLY
016               ;OR AS A CONSEQUENCE OF RESULTING RECURSIVE DEDUCTIONS
017
018               (SETQ TRACE NIL)
019               (COND ((NOT (EQ (GET NODE 'TRUTH) 'UNKNOWN))
020                      (ERROR 'SET-TRUTH--VALUE-NOT-UNKNOWN NODE)))
021               (PUTPROP NODE VALUE 'TRUTH)
022               (PUTPROP NODE SUPPORT 'SUPPORT)
023               (MAPC (FUNCTION (LAMBDA (CLAUSE) (PROG ()
024                                  (PUTPROP CLAUSE (1- (GET CLAUSE 'PSAT)) 'PSAT)
025                                  (COND ((= (GET CLAUSE 'PSAT) 0)
026                                         (SETQ CONTRA-LIST (CONS CLAUSE CONTRA-LIST))
027                                         (COND (CTRACE
028                                                (PRINT 'CONTRADICTION)
029                                                (PRINC (GET CLAUSE 'CLAUSE-LIST)))))))))
030                     (GET NODE (GET VALUE 'OP-CLAUSES)))
031               (COND ((GET NODE (GET VALUE 'EFFECT))
032                      (APPLY (GET NODE (GET VALUE 'EFFECT)) (LIST (GET NODE 'ASSERTION)))))
033
034               (MAPC (FUNCTION (LAMBDA (CLAUSE)
035                                (COND ((AND TRACE
036                                            (EQ CLAUSE CONTRA-SOURCE))
037                                       (ADD-2 CONTRA-CLAUSE 'CLAUSE-RESOLUTION)
038                                       (SETQ CONTRA-SOURCE NIL)
039                                       (SETQ TRACE NIL))
040                                      ((AND (= (GET CLAUSE 'PSAT) 0)
041                                            (NULL CONTRA-SOURCE))
042                                       (SETQ CONTRA-SOURCE CLAUSE)
043                                       (SETQ CONTRA-CLAUSE (MERGE CLAUSE NIL NODE NIL))
044                                       (SETQ TRACE 'TRUE))
045                                      ((SETQ F (DEDUCE-CHECK CLAUSE))
046                                       (SETQ TRACE 'TRUE)
047                                       (SETQ CONTRA-CLAUSE
048                                             (MERGE CLAUSE CONTRA-CLAUSE NODE (CAR F)))))))
049                     (GET NODE (GET VALUE 'OP-CLAUSES)))
050               (RETURN TRACE)))
051
052  (DEFUN DEDUCE-CHECK (CLAUSE)  ;THE FACT DEDUCED IS RETURNED ONLY IF A CONTRADICTION RESULTED.
053         (PROG (F)
054               (COND ((AND (= (GET CLAUSE 'PSAT) 1)
055                           (SETQ F (PCONSEQ CLAUSE))
056                           (SET-2 (CAR F) (CDR F) CLAUSE))
057                      (RETURN F))
058                     (T (RETURN NIL)))))
059
060  (DEFUN PCONSEQ (CLAUSE)
061         (DO CLIST (GET CLAUSE 'CLAUSE-LIST) (CDR CLIST)
062             (NULL CLIST)
063             (COND ((EQ (GET (CAAR CLIST) 'TRUTH) 'UNKNOWN)
064                    (RETURN (CAR CLIST))))))
065
066  (DEFUN MERGE (CLAUSE ACCUM EXCEPT1 EXCEPT2)
067       (PROG ()
068         (MAPC (FUNCTION (LAMBDA (NODE)
069                          (COND ((NOT (OR (EQ (CAR NODE) EXCEPT1)
070                                          (EQ (CAR NODE) EXCEPT2)
071                                          (MEMBER NODE ACCUM)))
072                                 (SETQ ACCUM (CONS NODE ACCUM))))))
073               (GET CLAUSE 'CLAUSE-LIST))
074         (RETURN ACCUM)))
```

```
001
002  ;TRUTH VALUE REMOVAL ROUTINES
003
004  (DEFUN REMOVE-TRUTH (NODE)
005         (PROG (REMOVED-LIST)
006               (REMOVE-2 NODE)
007
008               (MAPC (FUNCTION (LAMBDA (DOT)
009                       (PROG (NODE)
010                             (SETQ NODE (CAR DOT))
011                             (COND ((NOT (EQ (GET NODE 'TRUTH) 'UNKNOWN)) (RETURN T)))
012
013                             (NODE-DEDUCE-CHECK NODE 'TRUE)
014                             (NODE-DEDUCE-CHECK NODE 'FALSE))))
015                     REMOVED-LIST)
016
017               (MAPC (FUNCTION (LAMBDA (DOT)
018                       (PROG (NODE)
019                             (SETQ NODE (CAR DOT))
020                             (COND ((NOT (EQ (GET NODE 'TRUTH) 'UNKNOWN)) (RETURN T)))
021
022                             (COND ((GET NODE 'DEFAULT)
023                                    (PROG (CONTRA-SOURCE CONTRA-CLAUSE)
024                                          (PUTPROP NODE 'DEFAULT 'EXPLANATION)
025                                          (SET-2 NODE (GET NODE 'DEFAULT) 'PREMISE)))))))
026                     REMOVED-LIST)))
027
028  (DEFUN REMOVE-2 (NODE)
029         (PROG (VALUE)
030               (COND (VPRINT (PRINT '|REMOVE-VALUE |)
031                             (PRINC NODE)))
032               (SETQ VALUE (GET NODE 'TRUTH))
033               (COND ((EQ VALUE 'UNKNOWN)
034                      (ERROR 'REMOVE-VALUE--VALUE-NOT-PRESENT NODE)))
035               (PUTPROP NODE 'UNKNOWN 'TRUTH)
036               (PUTPROP NODE NIL 'SUPPORT)
037               (SETQ REMOVED-LIST (CONS (CONS NODE VALUE) REMOVED-LIST))
038               (MAPC (FUNCTION (LAMBDA (CLAUSE) (PROG ()
039                             (PUTPROP CLAUSE (1+ (GET CLAUSE 'PSAT)) 'PSAT)
040                             (COND ((= (GET CLAUSE 'PSAT) 1)
041                                    (SETQ CONTRA-LIST (DELQ CLAUSE CONTRA-LIST)))))))
042                     (GET NODE (GET VALUE 'OP-CLAUSES)))
043
044               (COND ((GET NODE 'MAKE-UNK)
045                      (APPLY (GET NODE 'MAKE-UNK) (LIST (GET NODE 'ASSERTION)))))
046
047               (MAPC (FUNCTION (LAMBDA (CLAUSE) (PROG (F)
048                             (COND ((AND (> (GET CLAUSE 'PSAT) 1)
049                                         (SETQ F (CAR (CONSEQ CLAUSE)))
050                                         (EQ CLAUSE (GET F 'SUPPORT)))
051                                    (REMOVE-2 F))))))
052                     (GET NODE (GET VALUE 'OP-CLAUSES)))))
053
054  (DEFUN CONSEQ (CLAUSE)
055         (DO CLIST (GET CLAUSE 'CLAUSE-LIST) (CDR CLIST)
056             (NULL CLIST)
057             (COND ((EQ (GET (CAAR CLIST) 'TRUTH) (CDAR CLIST))
058                    (RETURN (CAR CLIST))))))
059
060  (DEFUN NODE-DEDUCE-CHECK (NODE VALUE)
061         (PROG (CONTRA-SOURCE CONTRA-CLAUSE)
062               (DO C-LIST (GET NODE (GET VALUE 'CLAUSES)) (CDR C-LIST)
063                   (OR (NULL C-LIST)
064                       (EQ (GET NODE 'TRUTH) VALUE))
065                   (DEDUCE-CHECK (CAR C-LIST)))))
```

```
001
002   ;EXPLANATION AND BACKTRACKING ROUTINES
003
004   (DEFUN WHY (OBJECT)
005          (PROG (SUPPORT CLIST WHY-LIST)
006                (COND ((SETQ CLIST (GET OBJECT 'CLAUSE-LIST))
007                       (RETURN (MAPCAR (FUNCTION (LAMBDA (F) (CAR F))) CLIST)))
008                      (T
009                       (SETQ SUPPORT (GET OBJECT 'SUPPORT))
010                       (COND ((EQ SUPPORT 'PREMISE) (RETURN 'PREMISE))
011                             (T (DO CLIST (GET SUPPORT 'CLAUSE-LIST) (CDR CLIST)
012                                    (NULL CLIST)
013                                    (COND ((NOT (EQ (CAAR CLIST) OBJECT))
014                                           (SETQ WHY-LIST (CONS (CAAR CLIST) WHY-LIST)))))
015                                (RETURN WHY-LIST)))))))
016
017   (DEFUN BACKTRACK ()
018          (PROG (ASSUMPTION ASSUM-LIST)
019                (SETQ ASSUMPTION NIL)
020                (SETQ ASSUM-LIST NIL)
021                (COND ((NULL CONTRA-LIST)
022                       (RETURN T)))
023
024                (DO CONTRA CONTRA-LIST (CDR CONTRA)
025                    (OR (NOT (NULL ASSUM-LIST))
026                        (NULL CONTRA))
027
028                    (MAPC (FUNCTION (LAMBDA  (NODE) (FIND-ASSUM NODE 1)))
029                          (WHY (CAR CONTRA))))
030
031                (DO ((DO-ASSUM ASSUM-LIST (CDR DO-ASSUM))
032                     (MIND 1000 MIND))
033                    ((NULL DO-ASSUM))
034                    (COND ((< (CDAR DO-ASSUM) MIND)
035                           (SETQ MIND (CDAR DO-ASSUM))
036                           (SETQ ASSUMPTION (CAAR DO-ASSUM)))))
037
038                (COND ((NULL ASSUMPTION)
039                       (PRINT '|CONTRADICTION DEPENDS ON NO ASSUMPTIONS|)
040                       (RETURN NIL))
041                      (T (REMOVE-TRUTH ASSUMPTION)))
042
043                (BACKTRACK)))
044
045   (DEFUN FIND-ASSUM (NODE LEVEL)
046          (PROG (SUPPORT ASSC)
047                (SETQ SUPPORT (GET NODE 'SUPPORT))
048                (COND ((EQ SUPPORT 'PREMISE)
049                       (COND ((EQ (GET NODE 'EXPLANATION) 'DEFAULT)
050                              (COND ((SETQ ASSC (ASSOC NODE ASSUM-LIST))
051                                     (RPLACD ASSC (MAX LEVEL (CDR ASSC))))
052                                    (T (SETQ ASSUM-LIST (CONS (CONS NODE LEVEL) ASSUM-LIST)))))))
053                      (T (MAPC (FUNCTION (LAMBDA (NODE)(FIND-ASSUM NODE (1+ LEVEL))))
054                               (WHY NODE))))))
055
056   (DEFUN SATISFY (NODES)
057          (MAPC (FUNCTION (LAMBDA (NODE)
058                          (COND ((NOT (EQ (GET NODE 'TRUTH) 'UNKNOWN))
059                                 (PUTPROP NODE 'TRUE 'DEFAULT))
060                                (T (SET-TRUTH NODE 'TRUE 'DEFAULT)))))
061                NODES))
062
063   (TMS-INIT)
```

ADD-2	EXPR	002	008	FIND-ASSUM	EXPR	005	045	PCONSEQ	EXPR	003	060
ADD-CLAUSE	EXPR	002	004	MAKE-DEPENDENCY-NODE	EXPR	001	021	REMOVE-2	EXPR	004	028
BACKTRACK	EXPR	005	017	MERGE	EXPR	003	066	REMOVE-TRUTH	EXPR	004	004
CONSEQ	EXPR	004	054	NODE-DEDUCE-CHECK	EXPR	004	060	SATISFY	EXPR	005	056
DEDUCE-CHECK	EXPR	003	052	SET-2	EXPR	003	011	SET-TRUTH	EXPR	003	004
TMS-INIT	EXPR	001	006	WHY	EXPR	005	004				